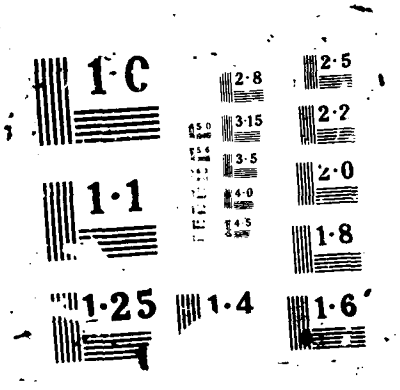
AD-A194 561 THE VIPER MICROPROCESSOR(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) J KERSHAW NOV 87 RSRE-87014 DRIC-BR-104861 1/1

UNCLASSIFIED F/G 12/6 NL

END
DATE
FILMED
8 88


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

BR10486

UNLIMITED

(2)

DTIC FILE COPY Report No. 87014

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

AD-A194 561

THE VIPER MICROPROCESSOR

Author: J Kershaw

[illegible]

UNLIMITED

TITLE: The VIPER Microprocessor

AUTHOR: J Kershaw

DATE: November 1987

SUMMARY

Most accidents are caused by human error. Computer control systems in aircraft, chemical plant, nuclear reactors and so on could in principle prevent many accidents, but in practice they are not reliable enough to be put in charge of human lives. This Report describes some of the developments in computer hardware and software which are needed before this situation can change, and introduces the VIPER microprocessor which has been designed specifically for ultra-reliable systems. In conjunction with a number of other RSRE Publications (see references) it defines the VIPER architecture formally and describes some of its supporting software.

The VIPER Microprocessor

J Kershaw

CONTENTS

1 THE NEED

Most accidents are caused by human error. The more reliable the engineering of aircraft, ships, nuclear power stations, and so on becomes, the more this is true: the majority of serious aircraft accidents are ascribed to human error [1], and the Three Mile Island incident might have been only a mishap if the operators had not over-ridden the automatic control system [2]. Most of these accidents would not have happened if totally reliable, trusted computers and totally correct software were available. The Chernobyl disaster is perhaps the exception: no safety system can do its job if it is switched off, but an effective system would not allow itself to be disabled while the reactor was running.

If computers could be trusted to control dangerous systems like aircraft and nuclear reactors, major gains in safety, performance, and economy could be made. Existing Flight Management Systems in civil airliners (which have only limited authority and are not regarded as safety-critical) are already yielding improved schedule-keeping and significant fuel savings.

Conventional hardware and software techniques are unlikely to inspire the level of confidence needed if computers are to be placed in charge of human lives. It is conceivable that computer hardware could achieve a failure rate of 1 in 10-to-the-9th hours operation, after prolonged separate testing of the components of a highly redundant system, but how could this be done for software? Software errors do not obey Gaussian statistics, which underly most reliability prediction: how can a software error be a random event, when it was present from the start?

A practical measure of confidence in a system is the cost of insuring it. When asked to insure an engineering structure, an underwriter will consider how complex it is, how near to its design limits it has been tested, how reliable its predecessors have been, and how costly any failure would be. On these criteria computer hardware is reasonably insurable, though most present-day processors are too complex for comfort.

Insurance of a complete computer system, including its software, against the consequences of breakdown is a much thornier problem. At least one death appears to have been caused by a fault in a computer program, controlling a hospital drug-dispensing machine. Who is responsible - the equipment manufacturer? The academic scientist who designed the algorithms? Or the programmer who expressed them in terms the microprocessor could understand? There is no clear-cut answer because, in this as in most cases, none of the steps in the design process was expressed in a sufficiently formal way for the error to be pinned down. The present state of the software art is simply not capable of achieving the degree of correctness demanded by life-critical systems.

2 THE HARDWARE PROBLEM

Modern computers are exceedingly complex. It is questionable whether any computer in general use has ever been fully specified, in the sense of allowing its response to every possible combination of inputs and instructions to be predicted. It is beyond question that none has ever been fully tested; an exhaustive test of even the simplest microprocessor would take billions of years.

Safety-critical systems are necessarily simple, and often need only limited computing power. Obviously, all the critical parts of such a system are replicated several times to guard against hardware failure. The moderate performance requirement and the need for numbers of every significant component both suggest use of single chip microprocessors. Unfortunately, existing microprocessors are highly unsuitable: they are very complex, not well specified, not generally available in "rugged" technologies, not guaranteed against future design changes, and inclined to ignore program errors such as arithmetic overflow which can have catastrophic effects. The complexity of these chips has other more subtle consequences: they are well beyond the reach of any formal validation technique, and they cannot make much use of redundancy to improve reliability because of the extra silicon area it would consume.

3 THE SOFTWARE PROBLEM

This will not be solved without a new programming technology. Though safety-critical programs are likely to be small, they are not so small that formal verification techniques can be applied directly to them. Beyond about 10000 machine instructions (tiny by modern standards) the work involved in formal, mathematical verification quickly becomes unmanageable. Even if a "proof" can be achieved, all that has been proved is that given hardware which exactly meets its specification the program will also exactly meet its specification. Either or both specifications may still be wrong. Most practical programs are therefore unverifiable, and likely to remain so.

The few life-critical programs which have been written so far have relied on rigorous programming discipline, prolonged testing, and (recently) mathematical analysis of the program texts. Two suites of programs developed originally at RSRE and Southampton University [3] are in regular use analysing the control flow, data use, information flow, and semantics of real-time software; they have found many errors and omissions but do not of themselves produce correct programs. They merely expose the errors in existing programs. Most of these programs are written in assembly language, but if a high-level language were used it might be possible to analyse both the source and the object program and to check for correspondence.

Programming discipline is much easier to enforce in a high-level language. Present-day languages such as Pascal contain constructs like variant records, computed "gotos", and pointer manipulation which though efficient could never be tolerated in safety-critical software. They may also allow range and bound checks to be "turned off" in the interests of speed. An ultra-reliable program cannot avoid these overheads; it must be written either in a strictly defined subset of a conventional language or in a special language. The result should be a highly-structured, easily analysed program, perhaps fairly bulky in source text form but usually quite small in object code.

In an ideal world, the designer of a reliable computer system would begin with:

(a) A formal specification of the requirement, provably complete, consistent, and unambiguous.

(b) A formal description of the programming language to be used, similarly provable.

Having chosen a computer, he would then have:

(c) A formal description of the hardware, and its response to every possible combination of inputs and instructions.

(d) A compiler to turn the language of (b) into machine code for (c), with a proof of correspondence between source and object programs for every construct of the language.

Once the programs are complete he will have:

(e) An operational program, with proofs that both source and object programs conform to (a).

The real world is rarely like this. (a) and (b) are practical in simple cases now, and a programming language is currently being developed in the Computing Division of RSRE which should be susceptible to formal definition [4].

(d) is only likely to be possible for a compiler which does very little optimisation, to avoid the problems of interaction between constructs.

(e) can be approximated by the program analysis techniques mentioned above, though a great deal of skilled human inspection is still needed.

Point (c), and the characteristics of a chip or chips which could make it possible, is the subject of the rest of this paper.

4 CHIP REQUIREMENTS

Most of the requirements below can be met individually by conventional microprocessor chips, though no existing chip can meet all of them. One of the most difficult requirements springs from the insistence of regulatory bodies such as the CAA and the Nuclear Installations Inspectorate that all the design and manufacturing details of safety-critical systems should be available to them; semiconductor manufacturers are very reluctant to disclose information of this kind, and are even more reluctant to commit themselves to long term supply of chips made by a specific process.

Most of the other requirements can be summarised in the single word "simplicity".

4.1 The chip must have a reasonably large address range, not to cope with large programs so much as to leave scope for multiple versions or "get-you-home" programs. 20 bits are probably enough.

4.2 The logic must be described in formal terms, e.g. in the hardware description language ELLA [9], and must be amenable to simulation with reasonable efficiency.

4.3 The design must be within the reach of a number of established semiconductor technologies, including those which can offer high temperature and radiation resistance.

4.4 The order code must be suitable for generation by a compiler, and it must incorporate permanent checks for a number of errors (e.g. arithmetic overflow) which are often ignored by conventional microprocessors. The order code must be simple and regular enough to permit analysis and (in special cases) verification at the object code level.

4.5 The interface to the outside world must be simple and predictable, since the chip will almost always be used with others of its kind in a multi-channel redundant system. This suggests that the chip should not have very much internal concurrency, a requirement already implied by 4.2 and 4.3.

4.6 The chip must have a reasonably high performance, at least 500000 operations per second.

4.7 The design process, from paper specification to silicon, should be as highly automated as possible. This minimises the risk of mistakes, gives the best chance of formal correspondence proving, and eases the task of implementing the chip in a range of technologies. A cell array or gate array technology is the most likely way of achieving this level of automation at present.

4.8 The chip should be easily testable. This implies that (at least) every memory element in the circuit should be readable and settable from outside in a limited number of cycles and ideally that every logic node should be accessible. The last is probably ruled out by the limited number of pins available on I.C. packages, but the first can be achieved by techniques such as "scan path" design, in which all the memory elements (flip-flops) in a chip are connected in a long serial shift register which can be clocked from outside.

4.9 The design must be to some extent "public". Manufacturers of systems will rarely use a device that cannot be obtained from at least two suppliers, and regulatory bodies like the CAA will not approve life-critical systems unless the whole design and manufacturing process is available for their inspection.

A chip to meet these requirements has been designed by the High Integrity Systems section of Computing Division and implemented in three distinct technologies by two British VLSI manufacturers. Its name is VIPER: a Verifiable Integrated Processor for Enhanced Reliability.

5 ARCHITECTURAL CONSTRAINTS

5.1 Word size

Floating point arithmetic is ruled out on grounds of complexity, and also because it is inherently inexact and therefore resistant to verification. This implies that VIPER systems must use fixed point arithmetic, which needs a lot of bits if neither range nor accuracy is to be sacrificed. 24 bits might just be enough for arithmetic, but it is an inconvenient size for instructions: too big for non-addressing instructions and too small (with a 20 bit address field) for memory addressing instructions.

Variable-sized instructions complicate the logic and the external interface severely, and also make the machine's behaviour in the presence of faults less predictable - consider the effect of obeying a program one byte out of alignment. Therefore VIPER instructions all occupy 32 bits, and all interface transactions are in units of 32 bits.

5.2 Memory management

Only the simplest memory management techniques are permissible in safety-critical software. Static, compile-time allocation is the rule, to avoid any risk of a program failing at run time through lack of memory. The VIPER architecture therefore does not make special provision for dynamic memory allocation, though it does not rule it out in case developments in analysis techniques make it acceptable in the future.

A simple stack, for subroutine entry and exit, is less of a risk. If recursion is absent it is easy to calculate the maximum subroutine nesting depth of a program, and static analysis of the code can show whether or not every path through a subroutine ends in a RETURN instruction (jumps out of subroutines are of course unimaginable). The VIPER architecture has no built-in stack, because of the extra logic it would require and because of a general desire to keep instructions free of side effects, but it allows a stack to be implemented easily in software.

Indirect addressing is avoided in reliable programs, except in the most rigidly constrained circumstances, but some means of computing an address is essential. VIPER allows indexed addressing, in which the contents of a register is added to the address field of an instruction to generate the eventual memory address, but also provides comparison instructions (see 5.3) which allow the index to be checked before use.

5.3 Arithmetic

VIPER provides 32 bit twos-complement addition and subtraction. Multiplication and division are not implemented in the present design but instruction codes have been left vacant for them.

Arithmetic overflow on either addition or subtraction causes the VIPER processor to stop, and signal to the outside world that an error has occurred. This is quite different from the response of a conventional microprocessor, which merely sets a flag and goes on to the next instruction. Unanticipated overflow is a catastrophic error, in that it delivers a grossly incorrect result, and the best response to errors in a safety-critical environment is usually to freeze and either allow the faulty channel to be out-voted (in a multi-channel system) or revert to a simpler back-up system. The very worst response is to go on processing and corrupt the rest of the system. Once stopped, a VIPER processor can only be restarted by the RESET signal.

There are situations where overflow is not significant, e.g. when adding or subtracting the less-significant parts of multi-precision numbers, and for these VIPER has different instructions. Unsigned add and subtract ignore overflow, but preserve "carry" from the 32nd bit in a single bit register called B so that it can be added to or subtracted from the next most significant part.

Yet another form of subtraction is needed for comparisons. This subtraction is clearly signed, but overflow is not serious because the result is only tested and not stored. The subtraction must be done, in effect, with a 33 bit arithmetic unit to ensure that the result always has the correct sign. The result of the VIPER "compare" instruction is a Boolean in the B register.

Apart from B, VIPER has no "flags" in the usual sense. Conventional microprocessors have 6 or more, and every one represents a conceptual doubling of the effort needed to verify a program and of the time needed to test a processor.

5.4 Interrupts

Random interrupts are undesirable in reliable software. Safety-critical programs do not normally allow interrupts, except possibly a timer interrupt at fixed intervals. Even this interrupt would probably be regarded as an error unless it occurred while the program was explicitly waiting for it. "Waiting for an interrupt" can be achieved just as well by polling, in which the program repeatedly asks the interrupting device whether it needs attention, with the advantage that polling is much less of an obstacle to analysis and/or verification than the possibility of random interrupts.

Any language designed for safety-critical applications will be constructed so that the maximum execution time of any sequence of statements can always be calculated: the density of poll instructions can be adjusted to give the required response time.

VIPER has no interrupt mechanism. A "reset" signal is provided to set the processor in a known state initially, but this is destructive: the previous machine state is lost.

5.5 Input/Output

In this VIPER is fairly conventional. Specific input and output instructions are provided, with the same address construction mechanisms as for memory references, but there is no reason why memory-mapped I/O should not also be used. All 20 address bits can be used to address peripherals.

The hardware interface is also conventional, with a REPLY signal so that slow peripherals can tell the processor when they are ready. One unusual feature is that the REPLY mechanism has a time-out: if REPLY is not asserted within 63 clock cycles after a request has been made, the processor stops and signals an error.

6 DESIGN AND MANUFACTURING CONSTRAINTS

VIPER could be made easily as a hand-crafted custom chip - it needs rather fewer gates than a Z80, in a much more regular structure. Building VIPER in this way would result in the highest performance and the smallest silicon area, but it would also require a massive design effort most of which would have to be repeated for each new manufacturing technology. Custom chips are also more difficult to simulate than cell array or gate array designs, where the characteristics of each primitive cell are known in detail. Repetitive arrays of cells are much more amenable to CAD than random logic networks, leading to a more highly automated and probably more predictable design process. A custom chip starts from scratch in accumulating reliability statistics, whereas a chip based on a widely used standard substrate can benefit from the experience of other devices. VIPER has therefore been based from the start on gate array and cell array technology.

The first VIPER implementation used the UK5000 gate array. This was a joint project between RSRE, Marconi Electronic Devices Ltd (MEDL), British Telecom, and a number of other UK electronics companies. "5000" refers to the number of "usable" gates on the device, allowing for the inevitably less than perfect efficiency of the layout process. The chip actually contains 410 dedicated single-bit latch circuits and 2400 general purpose gate cells, each made up of two N-channel and two P-channel MOSFET's. The latch cells are internally clocked, and are permanently connected together in a serial "scan path" register so that all of them are accessible for testing.

The manufacturing technology of UK5000 is silicon gate CMOS, with a minimum feature size of 2 microns approximately. Layout of a UK5000 chip is fully automatic, once a gate-level description exists. The CAD software is interfaced to the logic design language ELLA, so that a design can be taken automatically from a gate-level description in ELLA through to the final silicon.

VIPERS have been fabricated by MEDL on UK5000 substrates, and (as a private venture) using the CELLSOS Silicon-on-Sapphire process. The SOS chips are available commercially and can be supplied to various environmental requirements up to ESA "space qualified" standard. A third implementation is in progress at Ferranti Electronics Ltd using their proprietary CDI (bipolar) process, which offers good environmental performance with reasonably low cost. All three families of VIPER chips are pin-compatible.

The VIPER design is just practical on UK5000, with a bare minimum of "frills". Several highly desirable features have had to bc left out of the prototype design:

6.1 Multiplication and division. These must be done slowly by subroutine or by external peripheral hardware. Gaps have been left in the order code for them.

If VIPER had been designed as a microprogrammed machine, multiply and divide could be done with very simple hardware. Unfortunately a microprogrammed implementation would be much slower, and would need an external microprogram memory with its own special interface. This option was rejected in order to keep the basic machine simple and fast.

6.2 More registers. The present design has a minimal set consisting of:

An arithmetic register	A	(32 bits)
An addressing register	X	(32 bits)
A subroutine link register	Y	(32 bits)
An instruction address register	P	(20 bits)
The Boolean register	B	(1 bit)

These total 117 bits, and with various temporary registers use 220 latch cells and nearly all the combinatorial cells of UK5000. In practice X and Y can be used for arithmetic, and Y doubles as a second addressing register, but this duplication of function would be better avoided. Multiplication and division require an arithmetic extension register for 64 bit temporary products - again Y could be pressed into service.

6.3 Support for multi-channel redundancy. The initial VIPER chips ("VIPER 1") provide no special mechanisms for easing the interconnexion of groups of VIPERs in a multi-channel system. Theoretical work by Cambridge Consultants Ltd [14] has shown that fairly simple enhancements to the interface logic can allow a pair of VIPERs to work together as a unit which can detect and report virtually all faults in itself or in its shared memory. This new device is known as VIPER 1A, and is described briefly in Section 9.

6.4 Dynamic memory control. Safety-critical applications will probably avoid dynamic memory with its problems of periodic refreshing, radiation- and temperature sensitivity, and a higher random error rate than static memory. Ground test rigs however, and some less critical applications may be able to use the much cheaper and denser dynamic memory.

Periodic refreshing of dynamic memories can be done either by the processor (as in the Z80 and Z8000) or by external logic. An external refresh system needs its own timing, to enforce the maximum interval between refresh cycles, and a clash-resolution circuit to decide whether the processor or the refresh system asked first. This logic is complex and non-deterministic. A much better solution is to build the refresh control in to the processor, using the existing processor timing to avoid the need for clash resolution. Not enough cells are available in the UK5000 version of VIPER to accommodate this.

7 SUPPORT REQUIREMENTS

A surprising amount of supporting software and hardware is needed even to begin development of a microprocessor chip. Most of the items below would be needed for any major chip design, though the formal descriptions have all too frequently been skipped in the past. The items are described as far as possible in order of appearance.

7.1 An English description of the VIPER order code. This is included as Appendix 1.

7.2 A formal description of the order code in a language based on first order logic [6, 16]. The language used is LCF-LSM [5].

7.3 A high level simulation program to "animate" the design and allow early software testing. Our simulator is written in Algol 68 and runs on VAX machines; it can execute about 50 VIPER instructions per second [15]. In practice 7.2 and 7.3 have been developed together; the present simulator is as close as practicable to the LCF-LSM definition of VIPER and (interestingly) runs faster than its less rigorously constructed predecessor. The simulator is the practical definition of VIPER, and can be used to generate standard correct outputs from test programs. It may eventually be verifiable in the formal sense.

7.4 A simple assembly language for writing test programs. As its name (VITAL: VIper Temporary Assembly Language) implies, this language is not intended for applications programming.

7.5 A microprogrammed emulator to run VIPER programs at a realistic speed. The GEMINI microprogrammed host computer [7] has been used in conjunction with a CP/M-based microcomputer to provide a simple but practical VIPER programming environment. It runs at about 200000 VIPER instructions per second. The microprogram size (a measure of the complexity of the VIPER order code) is 168 GEMINI instructions. The complexity of the GEMINI architecture means that this microprogram is unlikely ever to be verifiable.

7.6 Comprehensive test programs to exercise 7.3 and 7.5, and to demonstrate correspondence between them. These are written in VITAL, and will continue to be developed throughout the project. The present suite takes about 20 minutes to run on the Algol 68 simulator. These programs are now used as acceptance tests for VIPER chips on delivery.

7.7 A logic design of VIPER in ELLA [8, 9]. Development of this overlapped items 7.2 to 7.6. The ELLA design was used both to produce one of the eventual chip layouts and to generate a set of test vectors for production-line testing of VIPER devices. Both manufacturers (MEDL and Ferranti) use these test vectors.

7.8 A host system to provide peripheral support for VIPER chips. This consists at present of a simple Single Board Computer with 8k x 32 bits of writable memory, a pair of buffered connectors to the VIPER bus, and a host processor (Intel 8039) to give control and loading facilities. A more powerful hardware development environment has been designed around a set of standard "Double Euro" cards, and will shortly be available commercially through Charter Technologies Ltd of Worcester.

7.9 A practical language for writing application programs. The obvious candidate is Pascal, of which a rigorously defined subset exists [13]. A compiler for this subset, generating VIPER code, is being developed and should be available by the Summer of 1988. Much work is being done on the use of Ada for critical software, and in the longer term there is no doubt that a reduced version of Ada will be the language of choice for use with VIPER. In the meantime, a language is required now. To fill the gap while suitable versions of Pascal and Ada are being developed, RSRE have implemented a structured assembly language called VISTA (VIper STructured Assembler) which, like the hardware development environment, will be available from Charter Technologies Ltd. A brief description of VISTA is given in Appendix 2.

8 VALIDATION OF THE DESIGN

VIPER is designed to be used in applications where a serious malfunction may result in loss of life or massive environmental damage. It is not enough therefore simply to produce a formal specification of the device; the chips themselves must be shown to conform to that specification in a way which satisfies Regulatory Authorities such as the Civil Aviation Authority and the Defence

Ordnance Board.

At the lowest level, actual devices can only be tested. "Proof" in the mathematical sense is not meaningful for a physical object which may "wear out" or be damaged. Two of the three VIPER implementations (UK5000 and Ferranti) contain extensive built-in test logic to improve test coverage; all VIPERs are subjected to the battery of tests described in 7.6 which have been generated directly from the design document.

This document must be correct. To ensure this is so, the design of VIPER (in ELLA) has been validated using formal algebraic techniques which have shown that the design implies the top-level specification - that is, given the design, the specification must be true. Much of the work of validation has been done at Cambridge by Cohn and Gordon [5, 10]. The whole validation process is described in references [6, 10, 11, 12, 16].

9 THE FUTURE

VIPER 1A chips (in Silicon-on-Sapphire) are expected to be available in the second quarter of 1988. These differ from VIPER 1 only in the interface arrangements; to the programmer they are identical. The two major enhancements are:

9.1 The 32 bit data bus is extended to include 8 parity bits. These are generated in such a way that any combination of memory errors within a single 8 bit byte is detected. Thus any breakdown of a single memory chip (e.g. loss of power supply) will cause a VIPER 1A to stop. In addition, more than 96% of random double-bit errors are detected.

Parity checking can be disabled for devices which are unable to generate the extra bits, e.g. simple peripherals.

9.2 VIPER 1A has a mode control input, which can set the chip into active or monitor mode. In active mode a VIPER 1A behaves much like a VIPER 1, and can be used singly in a non-critical application. In monitor mode, VIPER 1A does not generate outputs but instead inspects the input signals on what would have been the output pins. Internally it works as normal, comparing the signals it receives with their internally-generated counterparts. Any difference is signalled on the "Stop" pin which is cross-coupled to the "Error" input of the active chip.

To guard against faults in the comparator logic (which is used only in the monitoring chip) the essential components are duplicated. Each signal is kept in both normal and complemented form, using logic which delivers an illegal output in the presence of any single error.

A pair of VIPER 1A chips with shared memory and peripheral interfaces forms an error-detecting computer module which can then be used in a 2 or 3 channel fault-tolerant system. With careful electrical design, the error-detecting properties can be extended into memory and peripheral addressing - memory data errors are of course detected by the parity bits. Reliable error detection removes the need for voting, and with it much of the complexity of traditional fault-tolerant system designs; with paired VIPER 1As a simple active/standby architecture is nearly always adequate. Reference [14] gives more details of the system implications of VIPER 1A.

10 ACKNOWLEDGEMENTS

VIPER has been developed by the High Integrity Systems section of the Computing Division at RSRE. All the members of the section have contributed to practically every item mentioned in this Report. Much of the validation work, and the original development of LCF-LSM, were done by Dr M Gordon and Dr A Cohn at the Cambridge University Computing Laboratory. Cambridge Consultants Ltd have made a substantial contribution to the design of VIPER 1A.

11 REFERENCES

1. "World Airline Accident Summary"
Safety Data and Analysis Unit, Airworthiness Division, Civil Aviation Authority.

2. "Report of the President's Commission on the accident at Three Mile Island" by John G Kemeny. Library of Congress Catalog card number 79-25694, October 1979.

3. "The assessment of safety-critical software" by B D Bramson. MoD unpublished report, 1982.

4. "Orwellian programming in safety-critical systems" by I F Currie. Proceedings of the IFIP Working Conference on System Implementation Languages, experience and assessment. University of Kent at Canterbury, 1984.

5. "LCF-LSM" by Mike Gordon. Technical Report no. 41, University of Cambridge Computing Laboratory.

6. "VIPER Microprocessor: Formal Specification" by W J Cullyer. RSRE Report 85013, October 1985.

7. "GEMINI microprogrammer's handbook" by J Kershaw. RSRE Report 82015, 1982.

8. "Electrical, environmental, and timing specification of the VIPER microprocessor" by C H Pygott. RSRE Report 86006, June 1986.

9. "ELLA: a hardware description language" by J D Morison, N E Peeling, and T L Thorp. IEEE International Conference on Circuits and Computers, ICCC 82, New York, 1982.

10. "A proof of correctness of the VIPER microprocessor: the first level" by A Cohn. Proceedings of the Hardware Verification Workshop, University of Calgary, Canada, January 1987.

11. "VIPER - Correspondence between specification and major-state machine" by W J Cullyer. RSRE Report 86004, 1986.

12. "Formal proof of correspondence between the specification of a hardware module and its gate level implementation" by C H Pygott. RSRE Report 85012, November 1985.

13. "SPADE Pascal" by B A Carre and C W Debney. Program Validation Ltd, Southampton 1985.

14. "A self-checking computer module based on the VIPER micro-processor - a building block for reliable systems" by M P Halbert. Safety and Reliability Society Symposium, Altrincham, England, November 1987.

15. "VIPER: Simulation of Specification - User's Guide" by W J Cullyer. MoD unpublished report, 1986.

16. "Application of formal methods to the VIPER microprocessor" by W J Cullyer and C H Pygott. IEE Proceedings Vol 134 Part E, number 3, pp133-141. May 1987.

Appendix 1.

VIPER machine definition

The VIPER machine has 3 general purpose 32 bit registers (called A, X, and Y), a program address counter (P), and a single bit Boolean register (B). Memory addresses occupy 20 bits, so only the least significant 20 bits of P are meaningful: loading a "1" into any of the top 12 bits will cause the machine to stop.

All instructions occupy 32 bits, divided into a 12 bit function code and a 20 bit address.

```
| Function  12 bits  |        Address   20  bits          |
+--------------------+-----------------------------------+
```

The function code is further divided into:

```
|   RF    |   MF    |    DF     |   CF   |      FF      |
| 2 bits  | 2 bits  |  3 bits   | 1  bit |    4 bits    |
+---------+---------+-----------+--------+--------------+
```

Most instructions are of the form D := R op M, where D and R are registers chosen from A X Y P. M is either a 20 bit literal constant or the contents of a 32 bit memory location. Memory addresses are limited to 20 bits, and the machine will stop if a computed address (MF = 2 or 3, see below) exceeds this limit.

RF: source register field

0	R is contents of register A
1	R " " " " X
2	R " " " " Y
3	R " " " " P

MF: memory address control field

0	M is the address, i.e. 20 bit positive constant
1	M is [address], from/to memory or peripheral
2	M is [address + X] " " " "
3	M is [address + Y] " " " "

DF: destination control field

0	D is the A register
1	D " " X "
2	D " " Y "
3	D " " P "
4	D " " P " if B, else do nothing
5	D " " P " if NOT B, else do nothing
6	D is M in peripheral space) see
7	D is M in memory space) below

The sequence in which the CF, DF, and FF fields are inspected is important:

```
          |
          V
+-----------+   yes     The instruction is a comparison.  The
|  CF = 1   | ------>   DF field is ignored.
+-----------+
          | no
          V
+-----------+   yes     The instruction is  "store R":  M will
|  DF >= 6  | ------>   be used as the destination, stopping if
+-----------+           MF = 0.  The FF field is ignored.
          | no
          V
+-----------+   yes     If B is false do nothing, otherwise ALU
|  DF = 4   | ------>   operation to P.
+-----------+
          | no
          V
+-----------+   yes     If B is true do nothing, otherwise ALU
|  DF = 5   | ------>   operation to P.
+-----------+
          | no
          V
+-------------------------------------+
| ALU operation to register specified |
|    by DF (in the range 0 to 3).     |
+-------------------------------------+
```

ALU operations

If CF = 0 and DF < 6, FF and possibly MF specify the ALU function:

```
   FF = 0            D := -M  i.e. M operand complemented
        1            Y := P  then  P := M from memory space
        2            D := M from peripheral space  }  equivalent
        3            D := M from memory space      }   if M = 0
        4            D := R + M,  B := carry
        5            D := R + M,  stop on overflow
        6            D := R - M,  B := borrow
        7            D := R - M,  stop on overflow
        8            D := R XOR M
        9            D := R AND M
       10            D := R NOR M
       11            D := R AND -M

       12,  MF = 0   D := R/2, sign bit copied
            MF = 1   D := R >> 1 through B, i.e. D31 := B,
                                D0..30 := R1..31, B := R0
            MF = 2   D := R * 2, stop on overflow
            MF = 3   D := R << 1 through B, i.e. D0 := B,
                                D1..31 := R0..30, B := R31

       13            spare instruction, stop
       14            spare instruction, stop
       15            spare instruction, stop
```

Note that if the destination is P (DF = 3, 4, or 5) only functions 1, 3, 5, and 7 are legal. Attempting to obey any other will cause the machine to stop. Function 1 operates only on P; if DF specifies any other destination register the machine will stop.

The operation D := R can be achieved by performing D := R + 0 or (for destinations A, X, Y) D := R AND -0 which may be faster. D := -R can be done as D := R NOR 0.

Comparisons

If CF = 1, FF specifies a comparison. Comparisons never change anything other than B, apart from the change in P implied by continuing to the next instruction, and never cause the machine to stop unless a memory address exceeds 20 bits (MF = 2 or 3). For comparisons the ALU function is forced to "R - M", and the arithmetic is assumed to use at least 33 bits with initial sign extension and no overflow detection, allowing signed or unsigned comparison of 32 bit values.

The new value of B is derived from the result of the subtraction as follows - bit32 is the conceptual extra (33rd) bit:

FF = 0	B := bit32	(i.e. R < M)
1	B := NOT bit32	(R >= M)
2	B := allzero	(R = M)
3	B := NOT allzero	(R /= M)
4	B := bit32 OR allzero	(R <= M)
5	B := NOT bit32 AND NOT allzero	(R > M)
6	B := borrow	(unsigned R < M)
7	B := NOT borrow	(unsigned R >= M)

FF = 8 to 15: the set of operations is repeated, but if the result is 0 (i.e. FALSE) B is left unchanged.

Appendix 2

The VISTA Structured Assembler

VISTA is a high-level assembly language: its statements are essentially machine instructions embedded in an Algol 68 - like syntax. It will be superseded eventually by higher-level languages for application programs, but in the interim it provides a reasonably friendly programming environment and a vehicle for writing a limited range of VIPER application software.

VISTA is not a <u>safe</u> language, in the sense that SPADE Pascal is, though it does limit the programmer to structures which are reasonably free of complications. For example, there are no pointers or GOTO statements in VISTA. If VISTA is to be used for safety-critical software, the program texts must be subjected to static analysis by a tool such as MALPAS just as they would be with a conventional language. This process is made easy by MAVIS, a program which translates from VISTA to the MALPAS Intermediate Language preserving as much as possible of the programmer's original structure. In fact MAVIS and the VISTA language were designed together: VISTA includes only those constructs which can be analysed effectively by MALPAS.

This note describes VISTA informally by means of an annotated example, which uses most of the constructs of the language, though one or two have escaped e.g. CONTINUE. The program is fairly trivial and makes no pretensions to safety: it reads in a text and then counts occurrences of selected words in that text. The line numbers are merely for reference in the notes below, and are not part of the program.

The only aspects which will not seem reasonably familiar are the "region declarations" (lines 4 to 7) which define regions of memory into which code, constants, and so on are to be placed, and the fact that procedures in VISTA are declared after (rather than before) their calls. There is no profound reason for this: it simply seems more natural to put the main program first. No "loopholes" are allowed for recursion, which is forbidden in VISTA. Notice that A, X, Y are the VIPER machine registers.

```
1  PROGRAM  Example of VISTA: word frequency counter.
2
3
4  CODE  prog  FROM     0  TO  3999;          -- memory regions ..
5  CONST const FROM  4000  TO  4095;          -- .. with bounds
6  DATA  data  FROM  16r8000  TO  16r8000+8191;
7  PERI  peri  FROM     0  TO  1;
8
9
```

```
INT   last, count, match, k, p, q;       -- in the DATA region

CHAN  keyboard, screen;                  -- in PERI region

INT   key[20], text[9000];               -- more data, vectors

INT   sp = 32, eof = 26, cr = 13;        -- in the CONST region
INT   ms1[ ] = "\nInput the text: \0\";        -- messages
INT   ms2[ ] = "\nKeyword: \0\";
INT   ms3[ ] = "\nNumber of occurrences is \0\";
INT   ms4[ ] = "outside range\0\";

DISPLAY  Program starting!

A := 1;   CALL message;                  -- invite user to type ..
A := 1;   last := A;                     -- .. in a sample of text
A := sp;  text[0] := A;

WHILE  (CALL getchar; CALL upper)  A /= eof
DO  X := last;                           -- getchar may change X
    CASE  A >= 'A'  AND  A <= 'Z':  SKIP;        -- letters
          A >= '0'  AND  A <= '9':  SKIP         -- digits
    ELSE  A := sp                                -- all others
    ESAC;
    text[X] := A;  X := X + 1;           -- store the character
    last := X
OD;

X := last;  A := sp;  text[X] := A;      -- terminates last word

REPEAT  A := 2;  CALL message;           -- ask for a key word
        A := 0;  p := A;

        WHILE  (CALL getchar; CALL upper)  A /= cr
        DO  X := p;
            key[X] := A;                 -- read & store key word
            X := X + 1;
            p := X
        OD;

        X := p;  A := sp;  key[X] := A;  -- terminates key word
        IF  X = 0  THEN  BREAK  FI;      -- exit if null key word
        A := 0;  p := A;  count := A;

        WHILE  (Y := 0; match := Y; X := p)  X < last
        DO  WHILE  (A := key[Y])  A = text[X]
            DO  IF  A = sp
                THEN  A := 1;
                      match := A;        -- end of word, no mismatch
                      BREAK
                FI;
                X := X + 1;
                Y := Y + 1               -- more, keep trying
            OD;
            IF  (A := match; X := p; Y := text[X-1])
        A /= 0  AND  Y = sp
      THEN  A := count;              -- whole of key matched
            A := A + 1;              -- space before word?
            count := A               -- musn't find e.g. ..
      FI;                            -- .."king" in "smoking"
      X := X + 1;  p := X
    OD;

    A := 3;  CALL message;           -- print number of matches
    A := count;  CALL print

UNTIL  (A := key[0])  A = sp;        -- till null key word input
STOP;

-- Now for the procedures, notice they follow the main program

PROC  getchar:  (WHILE  (A := INPUT keyboard)  A = 0  DO  SKIP  OD);


PROC  upper:  (IF A >= 'a' AND A <= 'z' THEN A := A-('a'-'A') FI);


PROC  print:                         -- unsigned decimal output
 BEGIN  INT  lzflag;
  X := 0;                            -- suppress leading zeros
  lzflag := X;
  IF  A < 0  OR  A > 9999  THEN  A := 4;  CALL message
  ELSE  X := 1000;  CALL digit;
        X := 100;   CALL digit;
        X := 10;    CALL digit;      -- 4 digits maximum
        X := 1;     CALL digit
  FI;

  PROC  digit:                 -- prints 1 digit of number in A
   BEGIN  INT  power;          -- X must hold a power of 10, X /= 0
    power := X;
    Y := 0;
    WHILE  A >= power  DO  A := A - power;  Y := Y + 1  OD;
    IF  X = 1  OR  Y > 0  THEN  lzflag := X  FI;
    Y := Y + '0';                    -- convert to ASCII
    X := lzflag;
    IF  X = 0  THEN  Y := sp  FI;    -- leading zero
    OUTPUT Y, screen
   END
 END;    -- of "print"


PROC  message:                       -- message selected by ..
 BEGIN  INT  select;                 -- .. A register, 1 to 4
  select := A;
  X := 0;
  WHILE  TRUE                        -- i.e. do forever, exit..
  DO  CASE  (A := select)            -- .. by BREAK or RETURN
             A = 1:  (A := ms1[X]);
             A = 2:  (A := ms2[X]);  -- get 4 bytes from message
```

```
120                 A = 3:  (A := ms3[X]);
121                 A = 4:  (A := ms4[X])
122          ELSE   A := 0
123          ESAC;
124
125          Y := A AND 255;                      -- now output the chars ..
126          IF  Y = 0  THEN  RETURN  FI;         -- .. terminating on a NULL
127          OUTPUT Y, screen;
128          CALL right8;
129          IF  Y = 0  THEN  RETURN  FI;         -- return from "message"
130          OUTPUT Y, screen;
131          CALL right8;
132          IF  Y = 0  THEN  RETURN  FI;
133          OUTPUT Y, screen;
134          CALL right8;
135          IF  Y = 0  THEN  RETURN  FI;
136          OUTPUT Y, screen;
137          X := X + 1
138      OD;
139
140      PROC  right8:                            -- 8 place right shift
141        BEGIN
142          A := A/2;  A := A/2;  A := A/2;  A := A/2;
143          A := A/2;  A := A/2;  A := A/2;  A := A/2;
144          Y := A AND 255
145        END
146
147    END       -- of "message"
148
149  FINISH      -- of program
```

Notes (by line number)

1 All programs begin with PROGRAM, the rest of the line is taken as a title. VISTA has no "separate compilation" facility, but source-text inclusion is provided:

```
INCLUDE  [user.sub]file.vis
```

will incorporate the contents of the file named at the point where the INCLUDE directive appears. The file name follows standard VMS conventions. INCLUDE directives should occupy a line to themselves. An INCLUDEd file may itself contain INCLUDE directives, nested to any reasonable depth.

4..7 Region declarations. All storage in a VISTA program is allocated in a defined region, in ascending order of address. The assembler will report an error if a region fills up. Constants are kept separate from data to allow use of ROM. The PERI region has a separate address space from the other three. Notice the use of decimal and hex numbers (other bases can be used e.g. 8r for octal, 2r for binary) and of constant expressions which can be arbitrarily complex.

Notice the comment, beginning with -- and extending to the end of the line.

10 Ordinary "object" declarations in the DATA region. The only data types allowed are INT and BITS: both are simple 32 bit words. There is no restriction on the use of either, but future versions of the assembler may comment on suspicious usages e.g. logical operations on an INT. All storage allocation in VISTA is static.

12 Peripheral addresses in the PERI region, usable only with INPUT or OUTPUT (see lines 81, 107).

14 Vectors of variables, lower bound 0. Only one dimension is allowed.

16 Constants, to be placed in the CONST region. These are just ASCII character values. Wherever VISTA expects a constant (e.g. here or as a vector size, line 14) a constant expression can be written using any of the usual arithmetic and logical operators. Once declared, a named constant (e.g. "eof") can be used in constant expressions.

17..20 Vectors of constants. These can be given values either as a list of constant expressions e.g. (1, 2, 3, 4) or (as here) a string of 7 bit ASCII characters. Characters are stored four per word, least significant first. Notice the special characters \n (line feed) and \0\ (character of value 0, used here as a string terminator). Characters can also be used as individual constants: the declaration on line 16 could equally well be written as

```
INT  sp = ' ',  eof = '\26\',  cr = '\c';
```

22 DISPLAY is treated as a comment, but the text which follows (up to the end of the line) is included in the output file for use by the VISOR simulator program. VISOR will print the "display" text whenever the instruction following is obeyed.

24 et seq. Instructions are translated one-for-one to VIPER machine instructions. There are three 32 bit registers called A, X, Y of which X and Y can be used as index registers. Y is changed by the CALL instruction, but otherwise the registers can be used interchangeably.

28 Notice the "preface block" (CALL getchar; CALL upper) which is optional before a condition to set up the register contents. No explicit arguments are allowed with procedure calls, but A and X can be used freely to hand over parameters.

30..33 The VISTA "CASE" statement requires an explicit predicate on each limb. A single preface block is allowed before the first condition. Conditions are tested in the order written and never cause a change in the register values; when a condition is found to be true the corresponding limb is entered and followed by exit from the CASE statement.

Notice the use of AND as a connective in conditionals. Any number of AND or OR connectives may be used, but not both in the same condition. No brackets are allowed. Without these restrictions (which arise because of the need to map VISTA statements into single VIPER instructions) it would have been possible to use a simple IF - THEN - ELSE here.

Character constants generate the standard ASCII representation without parity.

The SKIP statement has no effect (and generates no code) but is necessary for syntactic reasons.

32 The ELSE limb is optional. If it were omitted and no condition was true, a CASE statement would cause the machine to stop. If you want a CASE statement to do nothing in this situation, use ELSE SKIP.

40 Loop constructs are WHILE - DO - OD and REPEAT - UNTIL.

51 BREAK and CONTINUE are valid only in a WHILE or REPEAT statement. Their effect is, respectively:

> Exit from the most local loop, i.e. jump to the statement following OD or UNTIL <condition>
>
> Proceed to the next cycle of the most local loop, i.e. jump to the beginning of the condition after WHILE or UNTIL.

The jump in this case is to line 77

77 The STOP statement generates an illegal VIPER instruction which stops the processor.

81 Procedure declarations in VISTA follow their calls, i.e. only forward reference is allowed. This is logically more satisfactory (the main program comes first) and matches MALPAS better than the traditional rule of declaration before use. Recursion is not possible, since the language does not allow pointers.

Variables and constants (lines 10..20) must be declared before use in the conventional way. In this and most other respects VISTA conforms to the normal Algol or Pascal block structure. Brackets are interchangeable with BEGIN .. END but must be correctly paired.

The peripheral location "keyboard" is assumed to deliver 0 until a character is typed.

84 "upper" turns lower case letters in A to upper case.

87 4 digit unsigned decimal output.

98 "digit" is nested within "print".

102 Division by repeated subtraction, since VIPER has (at present) no divide instruction. The loop is never obeyed more than 9 times.

107 The peripheral location "screen" is assumed to make characters visible in some way.

116 WHILE TRUE gives an endless loop which can be left only by BREAK or (in a procedure) RETURN.

117 to 122 The strings output by "message" must be known to the procedure in advance. It is not possible to hand over the address of an arbitrary string since VISTA does not allow pointers.

126 A zero character terminates the string. Strings are stored with the least-significant 8 bits of the word holding the first character.

140 "right8" is nested inside "message".

142 These are "arithmetic" shifts which duplicate the sign bit. Line 143 renders this unimportant. The alternative is an end-around shift through B (e.g. A >> 1) which in this case would do just as well.

149 Every program ends with FINISH

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Report 87014	3. Agency Reference	4. Report Security Classification Unclassified
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title THE VIPER MICROPROCESSOR
7a. Title in Foreign Language (in the case of translations)
7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Kershaw J	9(a) Author 2	9(b) Authors 3,4...	10. Date 1987.11	pp. ref. 26

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement
Descriptors (or keywords) continue on separate piece of paper

Abstract

Most accidents are caused by human error. Computer control systems in aircraft, chemical plant, nuclear reactors and so on could in principle prevent many accidents, but in practice they are not reliable enough to be put in charge of human lives. This Report describes some of the developments in computer hardware and software which are needed before this situation can change, and introduces the VIPER microprocessor which has been designed specifically for ultra-reliable systems. In conjunction with a number of other RSRE Publications (see references) it defines the VIPER architecture formally and describes some of its supporting software.

S80/48

DATE

ILMED

8